Segue & ColorGuard: Optimizing SFI Performance and Scalability on Modern x86

Shravan Narayan[†‡] Tal Garfinkel[†] Evan Johnson[†] David Thien[†] Joey Rudek[†]
Michael LeMay[⋆] Anjo Vahldiek-Oberwagner[⋆] Dean Tullsen[†] Deian Stefan[†]
[†]UC San Diego [⋆]Intel Labs [‡]UT Austin

Abstract

WebAssembly (Wasm) and similar Software-based Fault Isolation (SFI) systems enable secure sandboxing by virtualizing process address space. They accomplish this by: (1) adding a base address to the operand of all load/store instructions to select a sandbox, and (2) enforcing isolation by trapping out-of-bounds memory accesses using regions of unmapped memory (guard regions). Leveraging modern x86 hardware, we offer two optimizations to this.

With Segue, we observe that x86-64 segmentation can be used to remove most of the cost of SFI base addition, resulting in speedups ranging from 13.8% for SPECint® 2006 to 11.2% for font rendering in Firefox.

With ColorGuard, we note that MPK-based page coloring can be used to reclaim the virtual address space wasted by guard regions. This results in a 11.91× *increase in the number of concurrent Wasm instances a process can support — reducing context switch overheads, load imbalances, and other inefficiencies that detract from the performance of high-scale edge computing platforms.*

1 Introduction

WebAssembly (Wasm) is an essential part of the software ecosystem. It runs on billions of browsers around the world, supporting applications ranging from Zoom [6] and Figma [8] to Photoshop [32] and Google Earth [31]. Firefox relies on it to sandbox untrusted native libraries [10, 23]. It enables safe extensibility in the data center via infrastructure like Istio [28] and server-side applications like Shopify [5]. Simultaneously, Wasm is enabling a new generation of high-scale, low-latency, function-as-a-service (FaaS) edge computing platforms such as Fastly's Compute@Edge [27] and Cloudflare's Workers [18].

Wasm relies on Software-based Fault Isolation (SFI) [35] to enforce memory isolation. We offer two optimizations — *Segue* and *ColorGuard* — that improve SFI performance and scalability.

The key to improving performance is reducing instrumentation overhead. SFI enforces isolation by instrumenting every memory operation in an application. These operations are often part of the application's critical path; thus, reducing instrumentation overhead in even small ways enhances performance in a wide range of workloads, such as those on the client-side web.

The key to improving scalability (i.e., how many instances we can run in a process) is tied to how Wasm is used server-side in high-scale FaaS platforms such as Cloudflare's and Fastly's. These systems spin up a new Wasm instance in response to every network request—and handle massive numbers of requests concurrently. Wasm's fast startup times (30 μs [26]) and low context switch overheads [19] (2-3 orders of magnitude cheaper than OS context switches [15]) are critical for enabling this. To maximize efficiency, FaaS providers want to minimize the number of OS-level worker processes needed to run Wasm instances—this reduces OS context switch overheads, prevents load imbalances, and enables efficient communication between instances [7].

This paper introduces Segue and ColorGuard to improve SFI performance and scalability by leveraging the unique features of modern x86 processors.

With *Segue*, we note that all SFI instrumentation has two steps. First, a base address (e.g., the start of a Wasm linear memory) is added to the operand of a memory operation, (e.g., an `i32.load` in Wasm). Second, a bounds check is performed. This is similar to a segmented memory system. It turns out that the x86-64 ISA retains a vestige of x86 segmentation support; thus, by moving our base address to a segment register and using segment relative addressing (e.g., as part of the x86 `mov` that implements an `i32.load`), we can eliminate at least one instruction, free up a general purpose register, and even free an operand slot in our x86 `mov` operation (that was previously used for base addition). This gives the compiler more freedom to efficiently allocate resources, resulting in speedups ranging from 5.4%-38% depending on the workload (see § 3).

With *ColorGuard*, we note that all production Wasm implementations rely on a large address space (4GB) and guard

```
// loads ptr + offset from the base of
  ↪ the Wasm heap
#define wasm_read64(ptr, offset) ...
#define wasm_read64(ptr)
  ↪ wasm_read64(ptr, 0)

u64 ptr = ...; // ptr as 64-bit int
// make ptr a 32-bit offset, then read
u64 a = wasm_read64(trunc32(ptr));
u32 arr = ...; // an array in the heap
u32 idx = ...; // index into the array
// access arr + idx
u64 b = wasm_read64(arr, idx);
```

(a) Example code snippet

```
; rax = wasm_base, rbx = ptr
; rcx = arr, rdx = idx

; trunc32(ptr)
mov ebx, ebx
; load wasm_base + ptr
mov r10, [rax + rbx]
; trunc32(arr + idx)
add ecx, edx
; load wasm_base + arr + idx
mov r11, [rax + rcx]
```

(b) Compiled w/o Segue

```
1 ; gs_base = wasm_base
2 ; rbx = ptr
3 ; rcx = arr, rdx = idx
4
5 ; load wasm_base + trunc32(ptr)
6 mov r10, [gs: ebx]
7 ; load wasm_base + trunc32(arr +
    ↪ idx)
8 mov r11, [gs: ecx + edx]
```

(c) Compiled w/Segue

Figure 1: Segue in Practice: *Our code snippet illustrates how two common code patterns can compile more efficiently with Segue, an integer-to-pointer conversion followed by a read, and a read through an array index. Without Segue we can see that each pattern takes two instructions, and consumes a general purpose register(*`rax`*) to store the linear memory base. With Segue each pattern takes a single instruction, and consumes one less register* `rax`*, and frees an operand slot (the instruction in (c) on line 6 only uses one register).*

region (4GB) per instance to enforce bounds without the need for explicit bounds checks [1, 2]. While this is fast, it also wastes large amounts of virtual address space and limits the total number of instances per process to roughly 16K. Using MPK-based page coloring, we can eliminate most of the guard regions and reduce address space sizes to closer to what applications actually consume. In FaaS workloads, where address space consumption of at most a few hundred megabytes is common, this offers a 11.91× increase in the number of instances we can pack into a single process.

We present these two techniques in the next section. In § 3 we evaluate the performance of these techniques in production Wasm toolchains. We survey some related work in § 4 and conclude in § 5.

2 Design

We briefly describe how SFI works in WebAssembly (§ 2.1), then explore how we optimize it with Segue (§ 2.2) and ColorGuard (§ 2.3).

2.1 SFI in WebAssembly

WebAssembly (Wasm) enables multiple isolated execution environments (sandboxes) within a single process address space. Memory isolation is enforced on 32-bit address spaces, called linear memories.

A Wasm load or store takes two 32-bit unsigned operands. A Wasm compiler will generate code to add these operands (resulting in a 33bit address), and add the result to a 64bit base address (the start of the linear memory for a particular sandbox), presumably perform a bounds check, then finally do the load or store with the computed address.

However, bounds checks are expensive [37], which is why they are rarely done in practice. Instead, production Wasm implementations generally enforce bounds implicitly using a system of large address spaces and guard regions—they combine a 4 GB linear memory address space and a following 4 GB unmapped memory region (a guard region). Thus, by construction, any 33bit unsigned offset plus the base address will be within 8GB of the base, and any access beyond the first 32-bit (4GB) address space will trap.

For the purpose of our discussion, two details here are important. First is the base addition step, which we largely optimize away with Segue. Second is the 8GB per-instance requirement, which we reduce with ColorGuard.

2.2 Reducing SFI overhead with Segue

Segue is an optimization for WebAssembly that leverages the partial support for segmentation in x86-64 CPUs. Historically, segmentation support started with the 8086 to expand memory beyond 16-bit addressing, was extended to support protection in the 80286, and grew to support 32-bit addressing [4].

Segmentation allowed applications to define segments — regions of memory identified by a start address (*the base*) and a size (*limit*) — for different portions of application memory such as the stack, heap, code, etc. Segmented applications expressed memory addresses as offsets into segments, relying on the hardware to convert each offset into a full global address (by adding the base), and to check that each offset is below the segment limit.

Segmentation, however, never found broad adoption in the most popular 32-bit operating systems. Instead, the large address spaces supported by 32-bit processors led to flat addressing being sufficient for a time. When applications finally did outgrow 32-bit addressing, the flat addressing model persisted

in x86-64. Support for segmentation was largely dropped in 64-bit mode. However, two segment registers (`%FS` and `%GS`) were enhanced in x86-64 to support 64-bit base addresses. Modern operating systems dedicate one of these registers to thread level storage (TLS), and the unused register (e.g., `%GS` on Linux) is free for other uses.

Segue leverages the available segment register to optimize base address addition in Wasm by storing the Wasm linear memory base in this register. Segue performs all Wasm linear memory accesses using segment-relative addressing[1].

Despite this hardware functionality existing in all x86-64-compatible CPUs for more than a decade, and the requirement for converting relative offsets in data structures for managed runtimes existing even longer, no Wasm runtime currently uses this approach. Prior 32-bit SFI implementations like vx32 [9] and NaCl [36] leveraged x86-32 segmentation to support efficient isolation—however, with the changes in x86-64, segmentation was no longer perceived to be useful beyond optimizing TLS. We offer the observation that this hardware is still beneficial today. We use Wasm as an initial example, but this technique could potentially generalize to other managed runtimes.

To see how Segue can benefit Wasm compilation, consider Figure 1. Figure 1a offers an example of two common ways a program might access memory. The first read (line 7 in Figure 1a) is a simple access into Wasm's heap from some previously calculated address. Looking at Figure 1b (which shows this code compiled without Segue), we can see the compiler must first use an instruction to truncate the 64-bit value `ptr` to 32 bits (Wasm heaps are a maximum of 4GB), and then use another instruction to add the Wasm linear memory base and access the memory. Back in Figure 1a, the second read on line 11 is accessing an array element, which similarly requires 2 instructions in Figure 1b.

If we instead compile with the Segue optimization (Figure 1c), we can see the following benefits:

- *Segue frees up a general purpose register (GPR)*, so it can be used for other computations. Today, Wasm compilers use a register to store the heap base. In Figure 1b we can see `rax` being used for this purpose, and how Segue uses the otherwise unused `gs` instead in Figure 1c.
- *Segue frees an operand slot in complex x86 memory addresses.* x86 supports a variety of memory address formats that can specify complex address calculations. Wasm compilers today must reserve one of these operand slots to specify the Wasm base as shown in lines 7 and 11 in Figure 1b (base is stored in register `rax`). Thus, the compiler cannot use this slot for other inline addition. In our example, we can see how this forces the Wasm compiler to use two separate instructions, first to add the values for `arr` and `idx`, and then separately perform the memory access (lines 9 and 11 in Figure 1b). Segue restores the operand slot and allows this operation in a single instruction(line 8 in Figure 1c).
- *Segue truncates segment offsets to enforce Wasm bounds.* Wasm compilers today must calculate machine addresses using 64-bit input values, because the Wasm linear memory base is 64 bits wide, and register operands of different sizes cannot be mixed in x86 memory operands. For example, in Figure 1a on line 7, we must use `rbx` (a 64-bit register) as the parameter when adding to `rax`. Thus, we must have a prior instruction for truncation (on line 5) to 32-bits (required because the Wasm address space is 4GB). With Segue, we use a particular x86 prefix to override the default address size and instead cause the segment offset to be limited to 32 bits. This truncation occurs within the `mov` instruction directly, as is apparent from its use of `ebx` in line 6 in Figure 1c.

These three benefits allow Wasm compilers to more efficiently use registers and emit instructions, which in turn speeds up computations. The only added cost is a slight increase in the size of memory instructions when using the `%GS` prefix and address size override prefix (x86 instructions are variable length). However, we believe in general the benefits outweigh this.

2.3 Improving Scalability with ColorGuard

While 64-bit address spaces may seem nearly unlimited, they are not. Intel® 64-bit CPUs only support a 48-bit address space[2], and only 47 bits are available to user space applications [17]. As each Wasm sandbox requires 8GB of address space, we can only create $2^{47}/8$GB sandboxes per process—roughly 16K.

Production FaaS services that use Wasm, such as Fastly, are already bumping up against this limit [26]. Running more sandboxes requires instantiating more processes. This increases overhead, as process context switches are expensive, and splitting Wasm instances across processes induces load imbalances and increases communication overheads between instances [7]. Unfortunately, this is only likely to get worse in the future. For example, the Wasm component model [34] is likely to significantly increase the number of desired instances for a given application. Thus, we must address the cause of this limit.

The main culprit is the memory footprint of Wasm sandboxes, which as already noted, demands 8GB for each instance, including the linear allocated space and guard page. This means, for example, that even if a sandbox uses only

[1] x86-64 allows specifying a prefix on memory instructions to indicate that the access is relative to the `%GS` segment. As of Ivy Bridge (2011), Intel® also added user space instructions to set the segment base without expensive system calls.

[2] While some high-end server-class Intel® CPUs support a 52/57-bit address spaces, this is available only in a small fraction of CPUs

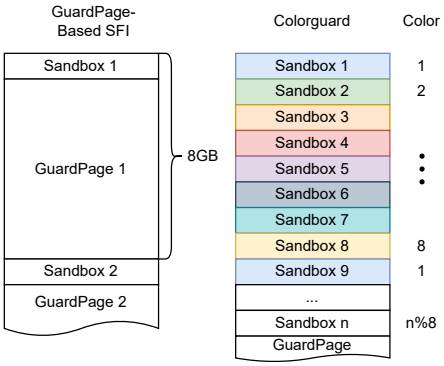

Figure 2: When using 1GB sandboxes, ColorGuard packs 8x as many sandboxes in the same region as traditional guard-page based SFI. Doing so is safe, since MPK enforces that sandboxes only access same-colored sandboxes, and ColorGuard ensures that same-colored sandboxes are 8GB away from each other.

1GB of memory, it still needs to reserve (but not necessarily commit) 8GB, wasting 78.5% of its assigned address space. In practice, Wasm instances in FaaS settings rarely exceed a few hundred megabytes [7].

Efficiently Packing Sandboxing with ColorGuard. To rectify this problem, we present ColorGuard, an optimization that efficiently packs sandboxes in memory by leveraging Memory Protection Keys (MPK) [17]. MPK is a recent x86 hardware feature [3] that adds thread-specific, hardware-enforced page permissions to the virtual memory systems.

MPK allows applications to assign a 4-bit tag (called a domain or color) to page table entries (PTEs) via system calls. The application can then manage which colors a thread has access to completely in userspace via the `pkru` register—`pkru` updates are very fast (roughly 25 cycles [25])—enabling rapid transitions between different protection mappings.

ColorGuard leverages MPK to increase the maximum number of sandboxes that can be run in an 8GB address range by a theoretical maximum of 15× (although in practice, we see a maximum of 11.91× as discussed in §3.2). The key intuition behind ColorGuard is that sandbox B's working memory can be used as sandbox's A's guard region, so long as A is a different color than B, ensuring that any access by A to B will trap.

To achieve this property, ColorGuard uses MPK to stripe memory such that nearby sandboxes have different colors. We see this illustrated in Figure 2 which contrasts the memory layouts of sandboxes using traditional guard regions with sandboxes using ColorGuard.

More precisely, ColorGuard requires every sandbox that occupies memory in the 8GB following the first sandbox use a different color for its linear memory; all other memory (including the memory belonging to the host application) is completely unchanged and assigned the default MPK color (0).

We see this illustrated in Figure 2. Here, each sandbox has a 1GB linear memory and uses one of eight MPK colors to stripe the 7GB region following the end of sandbox 1—offering an 8x increase in sandbox density. In our example, any out-of-bounds memory access from sandbox 1 would trap as it would hit a region with a different color. We could further increase density to 15×, by using all of MPK's colors and creating smaller sandboxes, i.e., for sandboxes of $8GB/15 \approx 550MB$.

Finally, we note that this striping pattern scales up to any number of sandbox chains — sandboxes that are placed in adjacent memory regions. We only need guard pages in a sandbox chain in two instances: (1) after the final sandbox to ensure the last sandbox in the chain is protected, and (2) if 15 consecutive sandboxes use less than 8 GB combined, we'll need a guard page before using the first color again. We also note that clever sandbox runtimes can also chain sandboxes of different sizes to efficiently use colors and possibly eliminate the second case; we leave such explorations to future work.

3 Evaluation

To evaluate the overhead of Segue and ColorGuard, we implement them in two Wasm compilers used in production today, then evaluate each mechanism on applicable benchmark suites.

Segue focuses on improving the performance of SFI; we thus benchmark it in the Wasm2c compiler [1] used by Firefox to efficiently sandbox buggy dependencies [16]. On the other hand, ColorGuard addresses scalability; we thus evaluate it in the Wasmtime compiler [2] used by cloud platforms like Fastly to sandbox computations from different clients in a cloud server.

Implementing Segue in Wasm2c. Wasm2c compiles Wasm by transpiling it to a limited subset of C which can then be compiled with a standard C compiler. We modified Wasm2c so that accesses to the heap were performed using a segment register. Specifically, our version of Wasm2c emits C code that relies on a GNU-extension called named address spaces [11]; this extension allows pointers in C to indicate that they belong to a particular segment. We then compile the emitted C code with Clang to get a native binary[4].

Implementing ColorGuard in Wasmtime. Implementing ColorGuard only required making three simple changes to the Wasmtime code to manage MPK domains. The first two changes (1) generate MPK keys on Wasmtime startup and (2) stripe memory by assigning these keys to alternating memory regions as described in Section 2.3. The third and final change

[3] MPK in this paper can refer to Page Protection Keys support, which was added to Intel® server-class systems as of Skylake (2017) and clients as of Tiger Lake (2020), or to Memory Protection Keys support that was added by AMD in EPYC Milan (2021).

[4] While GCC also supports the named address spaces extension, we observed that GCC's support was not robust, and using the extension often caused the compiler to crash during compilation.

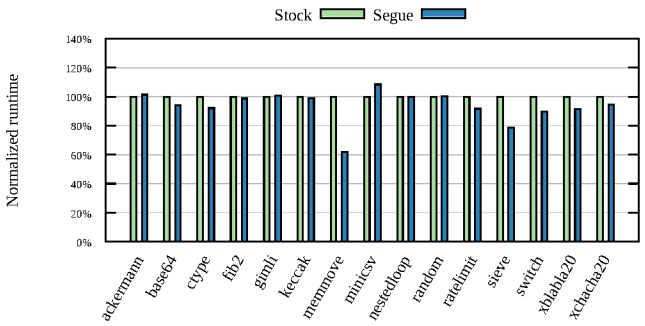

Figure 3: Sightglass performance of stock Wasm and Wasm with Segue. Segue speeds up computation by up to 38%, with median 5.4%. The one exception is the minicsv benchmark which incurs a slowdown of 8.4%.

is to update the MPK permissions prior to entering a sandbox so that the sandbox is only permitted to access its colors.

Setup. Segue benchmarks are run on an Intel Skylake i7-6700K (4 GHz) with 64 GB of RAM, running Ubuntu 20.04.5 LTS, with both frequency scaling and hyperthreading disabled. We also pin benchmarks to a single CPU that is isolated from other processes with CPU shield. ColorGuard benchmarks are run on an Intel Tigerlake i7-1165G7 (2.80GHz) with 16GB of RAM, with MPK support and running Ubuntu 22.04.1 LTS.

3.1 Evaluating Segue

To evaluate the performance benefits of Segue, we test the performance with three benchmarks/benchmark suites:

- **SPECint® 2006** is a popular set of CPU performance benchmarks representing single-thread, computation-heavy execution. We use the subset of SPEC's integer benchmarks which are Wasm-compatible (following Narayan et. al [24]). Notably, we opt for SPEC CPU® 2006 over SPEC CPU® 2017 due to the latter's increased memory requirements that regularly exceed Wasm's 33-bit address space.
- **Sightglass** is a suite used by members of the Bytecode Alliance, an organization that develops standards and tools for Wasm. It contains various short, "black-boxed" cryptographic, mathematical, and general-purpose programs typically executed in a WebAssembly environment.
- **Firefox's font rendering** is performed using an untrusted library, `libgraphite` [13], which is sandboxed using Wasm to ensure that any memory safety errors are contained. We measure the performance of font rendering by recording the time taken to reflow text on a webpage ten times with different sizes.

Analysis. Across all benchmarks, Segue tends to offer marked speedups over the default guard page mechanism. In the long-running SPEC benchmarks, Segue's median speedup is 7.8%;

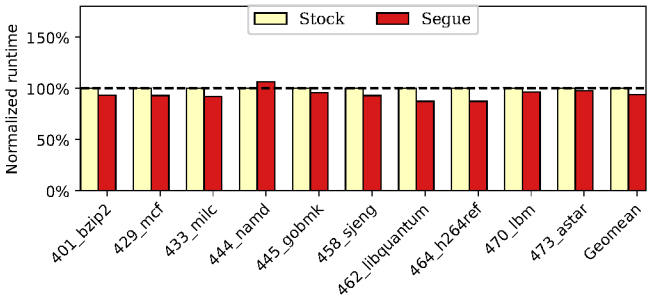

Figure 4: SPEC CPU® 2006 performance of stock Wasm and Wasm with Segue. Segue speeds up computation by up to 13.8%, with median 7.8% and geometric mean 6%. The one exception is the 444_namd benchmark which incurs a slowdown of 6.3%.

	Stock	Segue
Font render	2263 ms	2010 ms

Table 1: Firefox font rendering performance when compiling `libgraphite` to stock Wasm and Wasm with Segue. Segue speeds up computation by 11.2%.

in Sightglass, we see a 5.4% median speedup; and in the Firefox font rendering benchmark, we see a 11.2% speedup. We also observe from Table 2, that the binary size of the SPEC CPU® 2006 benchmarks are a median 7.1% smaller when using Segue. This is because of the more efficient code generation patterns discussed in §2.2.

We note that in both SPEC and Sightglass, one benchmark (`444.namd` and `minicsv`, respectively) incurs a slowdown (of 6.3% and 8.4%) when using Segue. We believe this could be because of the increase in memory instruction size to specify the use of `%GS` via Intel® prefix mechanism. While Table 2 indicates that Segue generates smaller code overall, the same fact may not hold true when measuring a few instructions in the tight inner loops of these benchmarks. If this expanded instruction size (and resulting instruction cache degradation) is not balanced by the other performance wins of Segue, the result is some performance overhead. To avoid these overheads, one could modify the compiler to choose between the Segue approach and an explicit base addition using a cost function. We leave this to future work.

3.2 ColorGuard

To evaluate the scalability benefits of ColorGuard, we create a custom benchmark that tests scaling of sandbox creation in the Wasmtime compiler and runtime. This benchmark simulates the high-scaling environment of an edge platform which must spawn tens of thousands of small sandboxes in a single process to sandbox computations from different clients [26]. Specifically, our benchmark instantiates 1024 512 MB sandboxes at a time until the runtime (using either traditional guard pages or ColorGuard) fails to `mmap` space for a new sandbox. We find that the upstream Wasmtime implementation that uses

	Stock	Segue	Binary size reduction
bzip2	356 KB	336 KB	6.0%
mcf	280 KB	272 KB	2.9%
milc	540 KB	504 KB	7.1%
namd	844 KB	780 KB	8.2%
gobmk	4464 KB	4272 KB	4.5%
sjeng	532 KB	496 KB	7.3%
libquantum	336 KB	324 KB	3.7%
h264ref	1352 KB	1200 KB	12.7%
lbm	280 KB	272 KB	2.9%
astar	404 KB	380 KB	6.3%

Table 2: Compiled binary sizes of the SPEC benchmarks comparing stock Wasm and Wasm with Segue. Segue decreases binary size by a median of 7.1%.

guard pages can allocate 21504 sandboxes before exhausting the address space. Using ColorGuard, Wasmtime is instead able to instantiate 256000 sandboxes, an 11.91× increase in address space density.

4 Related Work

The designs of Segue and ColorGuard draw upon a long history of SFI research.

SFI techniques and their cost The first instance of SFI was developed by Wahbe et al. [35]. Both Wahbe's implementation and subsequent work [9, 14, 22, 29, 30] incurred performance overheads between 20% and 30%. As a result, techniques have been developed to optimize these overheads [38] through static analysis. In contrast to these approaches, Segue removes overheads by leveraging hardware.

Various SFI systems have used x86-32 segmentation for sandboxing on 32-bit platforms [9, 19, 36]. However, using 32-bit segmentation in modern systems is infeasible as it requires sharing a 4GB address space across all running sandboxes. In contrast, Segue demonstrates how we can extract the benefits of segmentation even on 64-bit platforms.

SFI systems have used other CPU features to improve performance as well, for example Intel® Memory Protection Extensions (MPX) [20], MPK [15, 33], Intel® protection rings [21], virtualization [3, 12, 15], and ARM's memory domains [39]. However, use of such hardware comes at a cost. MPX has overheads comparable to software implementations [20], and MPK is restricted to 15 domains when directly used to sandboxed code [33]. Meanwhile, while ring, virtualization and Memory Domains incur expensive context switches due to ring/privilege switches [17]. In contrast, Segue allows SFI compilers to leverage hardware without incurring such extra costs.

Reducing register pressure in SFI compilers Register pressure has been a recurring source of performance degradation in SFI systems. Wahbe et al.'s SFI implementation on MIPS and Alpha machines reserved four and five registers respectively out of the available 32. Reserving these registers increased pressure on the remaining registers, which caused up to a 10% penalty on their benchmarks. Subsequent work [22, 29, 30] optimized SFI schemes so that they only reserved one register. However, for platforms like x86 and x86-64 which support only 8 and 16 general purpose registers respectively, this still produces significant register pressure. Segue takes the final step in reducing register pressure and reduces the number of reserved general purpose registers to zero, allowing for more efficient SFI implementations.

Guard pages and scaling in SFI tools Guard pages have long been used in SFI systems, and they have long been a bottleneck for scaling. For example, Wahbe et al. [35] used guard regions to protect the stack, and NaCl64 [29] used 80GB guard regions to optimize heap accesses. NaCl64's 80GB guard pages eliminate bounds checks but cost the system its scalability: indeed, these limit it to only 255 sandboxes [23].

Limited hardware resources have also caused scalability problems in SFI systems. For example, SFI tools that solely rely on MPK [15, 33] to enforce memory isolation have trouble scaling due to the limited size of the `PKRU` register. Since the `PKRU` only supports 16 domains, these systems cannot use more than 16 sandboxes, or they suffer expensive domain evictions which require swapping out pkeys of all pages belonging to a memory region [25]. In contrast, ColorGuard *improves* scaling by combining classic SFI with MPK.

5 Conclusion

For decades, SFI was largely an academic curiosity. In the last five years, WebAssembly has made it a critical technology for the internet—used by billions of clients, and an increasing number of servers around the world.

However, there are still limitations of Wasm that both constrain existing users and represent barriers to even greater adoption: performance and scalability. Segue is a novel use of existing Intel® hardware (segmentation) to bring down the cost of an instrumented load to a single instruction (the same as non-Wasm, non-SFI code).

ColorGuard uses a more recent Intel® feature (MPK) to enable guarded sandbox instances to be packed more densely, in the best case by more than an order of magnitude.

Acknowledgements

This work was supported in part by a Sloan Research Fellowship; by the NSF under Grant Numbers CNS-2155235 and CAREER CNS-2048262; by gifts from Google and Intel; and by DARPA HARDEN under contract N66001-22-9-4017.

References

[1] wasm2c. https://github.com/WebAssembly/wabt/tree/master/wasm2c, 2018.

[2] Wasmtime. https://wasmtime.dev, 2021.

[3] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *OSDI*. USENIX, 2012.

[4] J. H. Crawford and P. P. Gelsinger. *Programming the 80386*. Sybex Books, 1987.

[5] Duncan Uszkay. How Shopify uses WebAssembly outside of the browser. https://shopify.engineering/shopify-webassembly, 2020.

[6] Dylan Schiemann. Zoom on web: WebAssembly SIMD, WebTransport, and WebCodecs. https://www.infoq.com/news/2020/08/zoom-web-chrome-apis/.

[7] Engineers at large (anonymized for submission). private communication.

[8] Evan Wallace. WebAssembly cut Figma's load time by 3x. https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/, 2017.

[9] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of USENIX ATC 2008*. USENIX, 2008.

[10] N. Froyd. Securing Firefox with WebAssembly. https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly/, 2020.

[11] Gcc. Named address spaces. https://gcc.gnu.org/onlinedocs/gcc-9.1.0/gcc/Named-Address-Spaces.html.

[12] N. Goonasekera, W. Caelli, and C. Fidge. LibVM: an architecture for shared library sandboxing. *Software: Practice and Experience*, 45(12), 2015.

[13] Graphite - A free and open rendering engine for complex scripts. http://scripts.sil.org/RenderingGraphite, 2012.

[14] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *PLDI*. ACM, 2017.

[15] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 2019.

[16] B. Holley. WebAssembly and back again: Fine-grained sandboxing in Firefox 95. https://hacks.mozilla.org/2021/12/webassembly-and-back-again-fine-grained-sandboxing-in-firefox-95/, Nov. 2021.

[17] Intel® 64 and IA-32 architectures software developer's manual, 2020.

[18] Kenton Varda. WebAssembly on Cloudflare Workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/, 2018.

[19] M. Kolosick, S. Narayan, C. Watt, M. LeMay, D. Garg, R. Jhala, and D. Stefan. Isolation without taxation: Near zero cost transitions for sfi. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, January 2022.

[20] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *EuroSys*. ACM, 2017.

[21] H. Lee, C. Song, and B. B. Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1441–1454, 2018.

[22] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Security*. USENIX, 2006.

[23] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting fine grain isolation in the Firefox renderer. In *SEC*. USENIX, 2020.

[24] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, et al. Swivel: Hardening {WebAssembly} against spectre. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1433–1450, 2021.

[25] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. libmpk: Software abstraction for intel memory protection keys (intel mpk). In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 241–254, 2019.

[26] Pat Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, 2019.

[27] Pat Hickey. How Fastly and the developer community are investing in the WebAssembly ecosystem. https://www.fastly.com/blog/how-fastly-and-developer-community-invest-in-webassembly-ecosystem, 2020.

[28] Pengyuan Bian. Istio and Envoy WebAssembly extensibility, one year on. https://istio.io/latest/blog/2021/wasm-progress/, 2021.

[29] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *Security*. USENIX, 2010.

[30] G. Tan. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends in Privacy and Security*, 1(3), 2017.

[31] Thomas Nattestad. WebAssembly brings Google Earth to more browsers. https://blog.chromium.org/2019/06/webassembly-brings-google-earth-to-more.html, 2019.

[32] N. Thomas Nattestad. Photoshop's journey to the web. https://web.dev/ps-on-the-web/, 2022.

[33] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Security*. USENIX, 2019.

[34] L. Wagner. Component model design and specification. https://github.com/WebAssembly/component-model.

[35] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*. ACM, 1993.

[36] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *S&P*. IEEE, 2009.

[37] A. Zakai. Wasmboxc: Simple, easy, and fast vm-less sandboxing. https://kripken.github.io/blog/wasm/2020/07/27/wasmboxc.html, 2020.

[38] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *CCS*, 2011.

[39] Y. Zhou, X. Wang, Y. Chen, and Z. Wang. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM, 2014.